Administration Guide

Kabira Transaction Platform

This document is intended to address your needs. If you have any suggestions for how it might be improved, or if you have problems with this or any other Kabira documentation, please send e-mail to pubs@kabira.com. Thank you! Feedback about this document should include the reference KTPAG-3.6 . Updated on May 21, 2013 .

Published May 21, 2013

Kabira

Important Information

Contents

List of Figures

1

Overview

This book describes how to administer Kabira applications, which are deployed on the Kabira Transaction Platform (KTP). Other books in this series include:

- KTP Developer's Guide
- KTP Language Reference
- KTP Testing Guide
- KTP Development Process Guide
- KTP Advanced Development Guide

This guide focuses on administration of specific platform features. A better overview for the management of deployed Kabira applications is provided by the guide *Managing Kabira Solutions*.

Topics covered in this manual

This manual describes switchadmin commands used for administering Kabira applications, automatic configuration, and security and logging features available at the administration level.

Be sure to refer to the guide *Managing Kabira Solutions* for more about overall solution administration.

2

KTP operation

This chapter discusses switchadmin — the KTP command-line administration interface — and the commands used to administer nodes and run Kabira applications.

Using the switchadmin utility

Use the switchadmin utility to install, start, stop, remove, control and query a KTP node. It uses *plugins* — known also as *command targets* — to implement its actions.

The switchadmin utility is located at:

```
$SW_HOME/distrib/kabira/kts/scripts/switchadmin
```

The switchadmin utility also represents the command-line interface to a running application; it allows you, for example, to load, version, and monitor endpoints, services, and subscribers.

In general, you execute the switchadmin utility with commands that take the following form:

```
switchadmin [global <param>=<value>...] <command> <target> [<param>=<value>...]
```

Consider, for example, the following command:

```
switchadmin install switch application=example_company/example_application
```

This installs a KTP node in a subdirectory of the current working directory (using default values for all unspecified parameters).

You can develop additional plugins to perform application-specific administrative tasks; refer to the Application Developer's Guide for more details on developing these plugins.

Every target implements the two commands `help` and `display`. The `help` command supplies detailed help for a target. The `display` command is used for monitoring; the syntax and semantics of this command are different for each target type.

The following table provides a summary of the most commonly used switchadmin targets:

Target	Commands
switch	display, install, start, stop, remove, control, list available applications, and query a KTP node
configuration	display, load, activate, deactivate, and remove configurations (e.g., flows, endpoints, and services)
service	display, start and stop services
endpoint	display, start, stop, lock, unlock endpoints as well as enable and disable tracing at an endpoint
runset	display, start and stop a runset
script	display and run a KTP script
sessionfactory	display, start, and stop sessionfactories
session	display and stop sessions
switchsubscriber	display, close, start, stop switchsubscribers

This list of targets is by no means exhaustive, and every application can extend the list by adding application-specific plugins. For a detailed explanation of each of the targets, see the targets' help topics.

Getting help

Each plugin supplies its own detailed help. You have access to the help for some targets before you install and start a node. The help for others is accessible only after node startup. Use the following command for a brief overview of the targets with help available before node startup, as well as general help on the switchadmin utility:

```
switchadmin help
```

You can get detailed help on specific targets with the following command:

```
switchadmin help <target>
```

After you have started a node, you can supply the node's adminport to access the help of targets available only after node startup. Use the following command to display a list of all the targets available on the running node:

```
switchadmin adminport=<adminport> help
```

You can get detailed help for one of these targets using this command:

```
switchadmin adminport=<adminport> help <target>
```

Administering nodes

Typically you administer KTP application nodes by performing the following activities:

- the section called “Installing a node” on page 5
- the section called “Starting a node” on page 7
- the section called “Stopping and removing a node” on page 7
- the section called “Loading and unloading deploy specifications” on page 8

- the section called “Loading and activating configurations” on page 9

The switchadmin target for administering nodes is `switch`. For detailed help on this target, its commands, and parameters, type:

```
switchadmin help switch
```

Installing a node

The command used to install a node is:

```
switchadmin install switch
 [ application=<value> ]
 [ locationcode=<value> ]
 [ configpath=<value> ]
 [ trustedhosts=<value> ]
 [ installpath=<value> ]
 [ nodename=<value> ]
 [ adminport=<value> ]
 [ memorysize=<value> ]
 [ deploy=<value> ]
 [ buildtype=<value> ]
 [ deploymacros=<value> ]
 [ verbosenode=<value> ]
```

The following table describes the most common parameters used when installing a node:

Parameter	Description
application	Typically, you need to specify an application, in the form: company/application However, this parameter defaults to: kabira/ktsshell The parameter is used to construct the configuration directory. It locates the application.data file that describes the complete set of deployed subsystems making up the application. For example, the command switchadmin install switch application=CompanyA/ApplicationA installs a node and uses the data in $SW_HOME/distrib/CompanyA/ApplicationA/include/application.data to locate all deploy specifications, components and configuration files that need to be loaded. The application.data file (see the section called “Application data” on page 6) determines dependent applications to be deployed on the node as well as all their configurations and components (but not their deploy specifications).
configpath	An optional parameter that specifies a location for the config directory that is different from the default location.
installpath	An optional parameter that tells KTP where to install the node.

Use `switchadmin help` for more information regarding parameters.

Application data All KTP application files are partitioned according to company and application:

```
$SW_HOME/distrib/<company>/<application>
```

There is a file called `application.data` that is required for every application. This file specifies the company to which the application belongs as well as the name of the application and any other applications that it is dependent upon.

The Kabira IDE generates this file automatically and copies it to the appropriate directory when the application is built:

```
$SW_HOME/distrib/<company>/<application>/include
```

Here is an example of this file:

```
company: CompanyA
application: ApplicationA
uses: CompanyB/ApplicationB kabira/kts
```

The `uses` statement specifies all subsystems (other applications) on which the application is dependent, separated by spaces.

The order of the uses clause is important. The uses clause must list applications from the highest to lowest order in the dependency. Thus in the above example:

`CompanyA/ApplicationA` -uses-> `CompanyB/CompanyB` -uses-> `kabira/kts`

To list all applications available, invoke the following command:

```
switchadmin list switch
```

KTP node installation sequence While installing a node, the switchadmin utility gathers a list of all the application subsystems which together comprise the application. This is done by first locating the file application.data, using the value of the `application` parameter supplied with the `load switch` command. Then for the application and all its subsystems, the switchadmin utility performs the following tasks:

1. KTP builds a list of all application subsystems that comprise the application.
2. The switchadmin utility then creates the configuration directory for the node.
3. KTP will then install the underlying KTP node and start it. This does not start any KTP components.
4. It finds and loads all KTP deploy specifications.

At this point, the KTP system components, such as the system coordinator and the administration component, have been started, and all KTP components are loaded but not running.

The switchadmin utility automatically loads application configuration files found in the following directory:

```
$SW_HOME/distrib/<company>/<application>/config
```

It copies directories and files to the node configuration directory.:

```
<configpath>/<company>/<application>/...
```

<configpath> is the value passed as the `configpath` parameter when installing the node (the default value is `cfg`).

KTP automatically loads deploy specifications found in the following directories:

```
$SW_HOME/distrib/<company>/<application>/deploy
$SW_HOME/distrib/<company>/<application>/required
```

Remote Installation The `install` switch can be used to install a node on a remote host provided that KTP exists on that remote host. Use the `hostname` global parameter to specify the name of the remote host and the `kabirahome` global parameter to specify the location of the KTP product installation.

Starting a node

The command used to start a node is:

```
switchadmin adminport=<adminport> start switch
 [ deploy=<value> ]
 [ maxretries=<value> ]
```

KTP node startup sequence When the node is started, the following events occur:

1. If the node has not already been started, this is now done. All KTP components are started now.

2. All application configuration files are loaded.

Configuration loading during KTP startup consists of two distinct phases. First, KTP determines the set of components that are active on the node and loads all component-specific configuration files for those components (component-specific configuration files are those provided in the component's `catalogs` and `plugins` subdirectories). Then, once all component-specific files are loaded, KTP loads all other configuration files for the applications defined on the node.

Within each configuration directory, files are loaded in sequence according to their filenames, sorted using the collating rules of the locale. All configuration file names must be prefixed with two digits (*NN*-), for example, `50-mySubscriber.kcs`; files without this prefix are ignored.

KTP performs a depth-first search when loading configurations during node startup, loading and activating all configuration files that it finds.

Once loaded, each file is moved to an “activated” subdirectory and renamed with a file timestamp.

Configuration loading during node startup is an atomic operation. If any configuration file fails to load or activate, all configuration which occured as part of the startup process is rolled back, and no configuration of the node takes place. This error condition also causes KTP start to fail. An error messsage indicates the name of the configuration file that encountered the error, and the nature of the error condition itself. The node administrator must correct the problematic file, then issue the KTP start command again.

Stopping and removing a node

Use the following command to stop a node:

```
switchadmin adminport=<adminport> stop switch
```

This performs an orderly shutdown of the engines on the node, though the KTP system engines remain operational. To completely remove a switch node, use the following command:

```
switchadmin adminport=<adminport> remove switch
```

This shuts down the KTP system engines and remove all files in `installpath`.

Loading and unloading deploy specifications

Frameworks, such as channels or elements of KTP itself, ship with deploy specifications that define their components. For example:

```
deploy ApplicationA
{
    export
    {
         component myComponent;
         component anotherComponent;
    };
};
```

These specifications do not define any engines. Applications then use these specifications to include framework components that they require.

Applications must provide their own deploy specifications that do two things:

- Define the engines for the application
- Import and use the framework deploy specs for the frameworks used by the application.

For example:

```
importPath=$(IMPORTPATH);
import CompanyA.kes;
deploy someName
{
    engine someEngine
    {
        autoStart=FALSE;
        buildType=$(BUILDTYPE);
        use ApplicationA;
    };
};
```

Application deploy specifications end in a ".kds" suffix by convention.

For details on deploy specifications, see the Application Developer's Guide.

Load individual deploy specifications using this command:

```
switchadmin adminport=<adminport> load switch deploy=<deploy spec>
```

Note that if the node has successfully started, any specifications that were found in the required directory would have been moved to the `activated` subdirectory beneath it. If any files have been placed here since then, however, they will be loaded.

Loading and activating configurations

Most configuration data is loaded automatically when a node is installed and started. However, you may have configuration data that you will need to load and activate with the switchadmin utility, for example, to configure additional SwitchSubscribers (see Chapter 4).

To load and activate configuration data, invoke the following the command:

```
switchadmin adminport=<adminport> loadactivate configuration source=<sourcepath>
```

Scheduled switchadmin commands

A KTP node may be configured to run switchadmin commands at scheduled calendar days and times. Commands may be configured to run once, or multiple times. This is done with a KCS configuration file containing the command, and when it should run. The output from the command is published as KTP event switchadmin::EventIdentifiers::ScheduledCommandExecution to event topic kabira.kts.administration.

Example configuration which runs the switchadmin display runtimetimestatistics statistics=memoryusage command every day at 12 PM:

```
configuration "scheduledcommand" version "1.0" type "scheduledcommand"
{
    configure switchadmin
    {
        configure ScheduledCommand
        {
            Configuration
            {
                target = "runtimestatistics";
                command = "display";
                parameterList =
                {
                    {
                        name = "statistics",
                        value = "memoryusage"
                    }
                };
                scheduleType = scheduler::Periodic;
                minute = "0";
                hour = "12";
                dayOfWeek = "*";
                dayOfMonth = "*";
                month = "*";
                year = "*";
                fireExpiredTaskOnRestart = scheduler::Disable;
            };
        };
    };
};
```

See the IDLos documentation for the scheduler package for a description of the scheduleType, minute, hour, dayOfWeek, dayOfMonth, month, year, and fireExpiredTaskOnRestart configuration values.

Special purpose switchadmin targets

There are several other useful, special-purpose switchadmin targets:

switchadmin debugger

You can invoke the Kabira model debugger using this command:

```
switchadmin run debugger engine=engine-name
```

For more details on this target, type:

```
switchadmin adminport=<adminport> help debugger
```

The debugger also has extensive built-in help in the style of the dbx debugger.

The debugger is useful for resolving problems in consumers or rules, both of which require modeled implementations. For example, you might try to determine why a particular rule never succeeds by setting a breakpoint just before the rule evaluation.

Refer to the debugger’s built-in help for further information.

switchadmin scripts

Scripts may be written to be executed using the switchadmin `script` plugin. The value of this approach lies in the fact that the script will have access to the runtime environment of the running node, particularly the KTP administrative port, the location of the node installpath, and the configpath for the node. These data are passed as parameters to the script. For more information, please use

```
switchadmin adminport=<port> help script
```

to display the online help.

An example run script that is `listevents`, which creates an HTML table that lists the events described in all of the installed catalogs, together with the corresponding topics. This script is executed with the following command:

```
switchadmin adminport=<adminport> run script name=listevents
```

switchadmin runset

A runset is a collection of scripts and configuration files that act together to perform application configuration, initialization and termination.

A runset is identified by a name. The set of configuration files and executable scripts which comprise a runset are copied from the application distrib area to a KTP node's `configpath` directory as part of KTP node install:

```
<configpath>/<company>/<application>/runsets/<runset name>
```

Each runset directory may contain the following:

- A config subdirectory, containing the configuration files for the runset.
- A scripts subdirectory, containing the executable scripts for the runset.

- A runset.info file, containing the description and other metadata for the runset.

Once KTP is started, all installed runsets may be accessed via the switchadmin `runset` target.

To display a list of installed runsets and their states, invoke the following command:

```
switchadmin adminport=<adminport> display runset
```

Runset invocation You can invoke a runset on a running node using the switchadmin commands:

```
switchadmin adminport=<adminport> start runset name=<runset name>
switchadmin adminport=<adminport> stop runset name=<runset name>
switchadmin adminport=<adminport> display runset name=<runset name>
```

When you invoke `start runset`, the runset configuration files are loaded and activated. Then the start function in each of the runset's scripts is invoked, in ascending numeric order of the script names. If the load and activation of a configuration file fails, all configurations already loaded are deactivated and removed, and the start command fails. Note that in this case, the runset scripts will not be called. Similarly, if any of the runset scripts exits with an error status, all configuration is deactivated and removed, and no further scripts are called.

Invoking `stop runset` deactivates and removes all runset configuration, and calls all the stop functions, but in descending numeric order.

The `display runset` command displays the state of the given runset, if the `name` parameter is given, or of all known runsets otherwise. The display output contains the following information for each runset:

- The runset name.
- A runset state indicator, either "Running" or "NotRunning".
- The runset description, if any.
- A listing of the active configurations for the runset, if any.

3

KTP security

This chapter describes the security features of KTP, providing an overview of the underlying security architecture and a description of the security-related aspects of KTP administration and operational control. The chapter also provides guidelines for developing secure Kabira applications.

KTP security architecture

KTP provides a set of security services which enable development, deployment and operation of secure applications. The major security services provided by KTP are:

- Authentication
- Access Control
- Audit Trail
- Secure Network Communications
- Cryptography Services

These services are provided by the KTP Runtime, and are used by components, node management tools, and plugins. The KTP Runtime itself uses these services to ensure secure administrative access to KTP nodes. KTP allows application developers and administrators to dynamically configure security policy for KTP nodes, using the standard KTP configuration service.

Authentication

KTP Authentication is based on the concept of principals, or users of the system. A Principal is uniquely identified by a name, and has a set of configurable attributes. These attributes include the principal's text credential, and the roles granted to the principal. Principals are configured on a KTP node by loading and activating a configuration file which defines the principal. The following example configuration file illustrates the definition of a principal:

```
configuration "fred" version "1.0" type "security"
{
 configure security
 {
  config Principals
  {
   Principal
   {
    name = "fred";
    textCredential = "dino";
    roles =
    {
     "switchadmin",
     "developer"
    };
   };
  };
 };
};
```

The above configuration defines a principal named “fred” with a text credential of “dino”. The configuration also specifies the set of roles granted to the principal, “switchadmin” and “developer”. These roles are used by the KTP runtime to determine the access privileges of the principal when accessing protected resources on the node. Note that all KTP configurations which configure security policy must be of type “security”.

The authentication process verifies that a given principal is defined on the node and validates the presented credentials. Once authentication succeeds, all work done in the current transaction will be done in the context of an authenticated session for that principal. Any access to secured types during that session will require that the principal have a granted role which has access privileges for the given type. Principal definitions may be updated, for example to change the text credential or the set of granted roles, by loading a new configuration with the updated values.

Principal configurations may omit the "textCredential" directive to avoid embedding credential data in configuration files. KTP automatically generates a random text credential value for the principal in this case. KTP applications may access the generated credential and replace it with another value via the KSSL component interface.

Trusted host-based authentication

KTP security supports the configuration of trusted hosts, which allows for expedited authentication of principals when the authentication request originates in a network connection from a configured trusted host. Authentication from a trusted host passes without consideration for credentials. That is, if the authentication request originates from a trusted host, KTP trusts the host-based authentication mechanism (e.g. UNIX login) to have verified the identity of the given principal.

A trusted host may be specified via a security configuration specification, as illustrated in the following example:

```
configuraion "myhosts" version "1.0" type "security"
{
 configure security
 {
  configure Hosts
  {
   config myHost
   {
    name = "host1";
```

```
   };
  };
 };
};
```

In the above example, the host “myHost” (a simple host name, not a fully qualified domain name) is set to be a trusted host.

Trusted hosts can also be specified for the `install switch` command with the `trustedhosts` parameter, which accepts a comma-separated list of trusted hosts, for example:

```
switchadmin ... install switch trustedhosts=host1,host2,host3 ...
```

X509-based user authentication

KTP also supports X509-based user authentication for switchadmin commands. This is a built-in facility that allows you to integrate a KTP application into an existing public key infrastructure (PKI) environment. To activate X509-based user authentication for a specific issuer certificate authority (CA), you must make the CA's credentials available to the Kabira application.

Copy the CA's credential to the Kabira application's `credentials` directory in the distribution area:

```
$SW_HOME/distrib/<company>/<application>/credentials
```

When the Kabira application is installed and started, all the CA credentials found in this directory are automatically configured as trusted CA credentials for authenticating user-supplied X509 credentials.

A user requiring X509-based authentication must supply the X509 certificate by way of the optional parameter `x509Credential` for every switchadmin command. This parameter should be the local path of a PEM-encoded X509v3 certificate file for the calling user. The file should also contain the encrypted private key for the given user, and the password parameter should specify the value needed to decrypt that private key.

```
switchadmin adminport=<adminport> x509Credential=<x509Credential path>
password=<password> <command> <target> ...
```

When the switchadmin command is invoked with the `x509Credential` parameter, KTP will try to authenticate the user's credentials using each of the configured trusted CA credentials, until successful or authentication fails for all available CA credentials.

Access Control

The KTP Access Control facility allows the configuration of role-based access control rules to be enforced when a given type is accessed. An access control rule essentially grants a set of privileges to a given role. The following example configuration file illustrates the definition of an access control rule:

```
configuraion "myrules" version "1.0" type "security"
{
 configure security
 {
  config AccessControl
```

```
  {
   Rule
   {
    name = "mycomponent::myifc::myop";
    locked = true;

    accessRules =
    {
     {
      roleName = "switchadmin";
      permission = Execute;
     },
     {
      roleName = "developer";
      permission = Execute;
     },
    };
   };
  };
 };
}
```

In the above example, the security configuration includes two directives. The “locked=true” rule specifies that the “myop()” operation cannot be executed via un-authenticated access. That is, it is “locked” to public access. The second directive, “accessRules = ...” grants execute permission on the “myop()” operation to the “administrator” and “developer” roles. Once this configuration is active, only principals who have the above roles will be able to execute the operation.

KTP looks for and enforces access control rules for a type when the type is accessed. In the example above, when the myop() operation is invoked, KTP will determine the identity of the currently authenticated principal, and match the set of roles defined for that principal with the access control rules defined for the “myop()” operation. If the active principal has the “administrator” or “developer” role granted, the access control check passes and the operation is invoked. If the principal does not have a role which has execute priveleges for the given operation, the access control check fails, and an access violation exception is thrown. Throwing an access violation exception has the effect of aborting the current transaction.

Refer also to the security chapter of the KTP Developer’s Guide.

Default KTP security policy

All KTP components include a default security policy configuration which is automatically loaded and activated at node startup time. The policy for each component defines access control rules which restrict access to administrative plugin interfaces implemented by the component. The default KTP security policy specifies a set of access control rules based on two roles, “switchadmin” and “switchmonitor”. The “switchadmin” role provides configuration and operational control over the switch node. The “switchmonitor” role provides access to display-type operations, which allow for examination and monitoring of the state of a switch node.

Adding and modifying security policy

KTP developers and administrators can define security policy via configuration specifications, which can be loaded automatically as part of KTP node startup, or explicitly via the switchadmin “configuration” target. KTP supports automatic loading of application-level configuration files, and component-specific configuration files.

KTP looks for application-level security configuration files under:

```
$SW_HOME/distrib/<company>/<application>/config/00-security
```

where <company> and <application> are the corresponding values of the application or one of the applications it is dependent on.

KTP looks for component-specific security configuration files under:

```
$SW_HOME/distrib/<company>/<application>/config/<component>/02-security
```

where <company> and <application> are the corresponding values for the application or one of the applications it is dependent on, and <component> is the component name that the security configuration file refers to.

Typically, application-level security configurations contain policy directives which apply to a whole node. Principal definitions and trusted host definitions, for example.

Component-specific security configurations should contain policy directives which refer only to that component. This would typically be access control rules and audit directives for types in the component.

A Kabira application can have security policy automatically loaded and activated by placing configuration specifications in the above directories. This ensures that security policy is in place as the node is started.

KTP administrators may update existing security policy, or add/remove policy directives on a running node via the switchadmin "configuration" target. That target allows new configurations to be loaded and activated, and existing policy configurations to be deactivated and removed.

Administrators can update existing security policy directives by using the versioning facility supported by KTP configuration. For example, the following security configuration specification updates the roles assigned to a principal:

```
configuration "fred" version "2.0" type "security"
{
 configure security
 {
  config Principals
  {
   Principal
   {
    roles =
    {
     "switchmonitor"
    };
   };
  };
 };
};
```

The above specification updates the roles granted to a principal named “fred”. All other properties of the principal - the credential, for example - are not modified. Note that updating the roles for a principal resets the roles to the new value, so to add a role to an existing set, the new value should contain the old and new roles.

Security and KTP node operational control

The KTP operational control architecture consists of the switchadmin client which issues commands to a KTP node. These commands are received and processed by a set of administrative plugins defined by the application, and by KTP itself. This section discusses the security implications of this architecture.

Administrative plugin security

KTP requires that all administrative plugins be secured. That is, all administrative plugins must have a loaded and activated security configuration which defines access control rules for all operations supported by the plugin.

KTP script-based plugins, because they lack modeled interfaces, require a different approach. KTP security policy is enforced by the KTP runtime, so it only takes effect once the plugin calls a secured operation. This means that script-based plugins should be designed to invoke a secured operation as early as possible, forcing early security enforcement.

The switchadmin role

The built-in KTP security policy assigns administrative privileges to principals via the `switchadmin` role. All operations which modify the operational behavior of a KTP node may only be executed by principals which have `switchadmin` role privileges. Operational control of a KTP node occurs via switchadmin and its associated plugins. Security policy is enforced on the KTP interface elements called internally by these scripts and/or plugins. Users should keep in mind the operational impact of security policy enforcement, and specify access control rules for administrative and configuration interfaces, rather than business transaction processing interfaces.

KTP automatically assigns the `switchadmin` role to the principal defined for the user who installs the KTP node (identified by the user's unix login name). This principal is therefore the sole administrator of the node when the node is first installed. Additional users may be granted administrator privileges by loading and activating security configurations which define principals for those users and grants the `switchadmin` role to those principals.

The switchmonitor role

The default KTP security policy assigns system monitor privileges to principals via the `switchmonitor` role. This role is granted execute permission to all display-type administrative plugin operations. Consequently, this role may be granted to principals who act as system monitor clients. Such principals will be able to display the state the of the KTP node, but will be unable to execute administrative operations which change the operational state of the node.

Administrative plugin security example

The following example illustrates the definition of a security configuration specification for a KTP administrative plugin. A KTP administrative plugin is represented by an interface which is derived from "switchadmin::SwitchPlugin", as in the following code excerpt:

```
package p
{
 interface MyPlugin : switchadmin::SwitchPlugin
 {
```

```
  void doIt();

  void display();
 };
};
```

The above administrative plugin supports two public operations, doIt() and display(), which must be secured. The following security configuration specification secures this administrative plugin:

```
configuration "mysecurity" version "1.0" type "security"
{
 configure security
 {
  config AccessControl
  {
   //
   // Administrative plugin access control policy:
   // -- lock all unauthenticated access
   // -- grant full access to switchadmin role
   // -- grant execute access to the display()
   // operation to the switchmonitor role
   //
   Rule
   {
    name = "p::MyPlugin";
    locked = true;
    lockAllElements = true;
    accessRules =
    {
     {
      roleName = "switchadmin";
      permission = AccessAllOperationsAndAttributes;
     },
     {
      roleName = "switchadmin";
      permission = Create;
     },
     {
      roleName = "switchadmin";
      permission = Select;
     },
     {
      roleName = "switchadmin";
      permission = Destroy;
     }
    };
   };
   Rule
   {
    name = "p::MyPlugin::display";
    accessRules =
    {
     {
      roleName = "switchmonitor";
      permission = Execute;
     }
    };
   };
  };
 };
};
```

In the above security configuration, the "AccessAllOperationsAndAttributes" permission is used to grant access to all operations and attributes in the plugin interface "p::MyPlugin" to the "switchadmin"

role. A separate rule grants excute permission for the display() operation to the "switchmonitor" role.

The switchadmin command-line tool and SSH

The switchadmin command-line tool provides administrative access to KTP nodes running on both the local and remote hosts. Some commands sent to nodes running on remote hosts are executed via Secure Shell (SSH) connections. SSH is a tool which provides authentication and confidentiality for network connections to remote hosts. SSH requires configuration of authentication keys for the user who executes the remote request. Authentication keys for SSH are generated using the "ssh-keygen" command-line tool. See the "ssh-keygen" manual page (run "man ssh-keygen" on a Unix host) for further information on how to generate authentication keys, and where the keys must be stored for SSH to make use of them.

The switchadmin security plugin

KTP contains a "security" plugin which supports the following commands:

```
switchadmin display security
 [type=principals|accesscontrol|audit|hosts]
 [name=name]
 [detailed=true|false]

switchadmin audit security
 [name=name]
```

The "display" command's output contains a description of the currently configured security policy on the node. The required [type] parameter is used to specify what policy elements should be displayed.

The "audit" command is used to carry out an audit of administrative plugin interfaces active on the node. KTP security policy requires that all administrative plugin operations are secured. That is, that access control rules must be configured for all administrative plugin operations.

The "audit security" command checks that access control rules are configured for all public operations supported by the given administrative plugin, if [name] is given, or all administrative plugins on the node, if no [name] parameter is provided. If the audit finds unsecured public operations in an administrative plugin, the command fails with an error message which describes the types which lack access control.

The "audit security" command is automatically run during KTP startup, to ensure that all administrative plugins that are part of the applications running on the node are fully secured. Administrators may run the command at any time, to ensure that any administrative plugins which were added to the node after it was started are secure, and to ensure that security policy for any existing administrative plugin has not been removed.

4

Event logging

This chapter deals with KTP events and configurable subscribers for logging events to files.

The KTP event service is a publish/subscribe service for system and application events. KTP events are used for system monitoring, activity logging, and error handling. This chapter covers the activity logging aspects of the KTP event service, and more specifically configurable KTP event subscribers as well as the topics to which they can subscribe. (For more details on the KTP event service, see the Application Developer's Guide.)

Topics

Events are organized according to topics. The topics are a tree structure. Every event is published to a topic, and subscribers subscribe to one or more topics. Subscribers to a particular event topic receive all events published on that topic and any of its subtopics.

The following is an example topic hierarchy:

```
kts
kts.tracing
kts.tracing.flows
kts.tracing.endpoints
kts.tracing.debug
kts.alarm
kts.alarm.network
kts.alarm.content
```

A subscriber to `kts.alarm` sees any event published to `kts.alarm`, `kts.alarm.network`, and `kts.alarm.content`.

Administering SwitchSubscribers

KTP provides a ready-made subscriber, SwitchSubscriber, that requires only configuration to use. SwitchSubscribers write all received events to a log file.

There is a default SwitchSubscriber (`KTSSubscriber`) that subscribes to "kts" and thus listens to all KTP events. The default `KTSSubscriber` logs all events to:

```
kts_<nodename>_<timestamp>_<count>.log
```

Other SwitchSubscribers may be created automatically during node load and startup (see the section called “Administering nodes” on page 4). You can create and version any number SwitchSubscribers. You manage SwitchSubscribers from the command line with the swithadmin utility.

You can create, modify and delete SwitchSubscribers by loading, activating, and deactivating configuration data. Use the switchadmin `configuration` target.

You can start, stop, and display a SwitchSubscriber with the `switchsubscriber` target.

When a SwitchSubscriber is activated, it is automatically enabled. In this state it will log all events that it receives to the configured file.

A SwitchSubscriber is disabled when an i/o error occurs or when it is explicitly stopped with the following command:

```
switchadmin adminport=<port> stop switchsubscriber subscriberName=<subscribername>
```

When a SwitchSubscriber is disabled, it will drop any events it receives without logging them to file.

A disabled SwitchSubscriber is enabled with the following command:

```
switchadmin adminport=<port> start switchsubscriber subscriberName=<subscribername>
```

To display the state of a SwitchSubscriber and the path to the configured log file, use the following command:

```
switchadmin adminport=<port> display switchsubscriber subscriberName=<subscribername>
```

Without the `subscriberName` parameter the `display` command presents the state and log file path of all SwitchSubscribers.

For detailed help on the switchadmin `switchsubscriber` target, type

```
switchadmin adminport=<adminport> help switchsubscriber
```

Configuring SwitchSubscribers

A SwitchSubscriber configuration controls many aspects of the event subscriber's behavior, including:

- the topics subscribed to
- the file to which the events are written
- the format of the event text
- the rollover policy for the output files

The following is an example SwitchSubscriber configuration:

```
configuration "MySubscriberConfig" version "1.0" type "Subscriber"
{
   configure switchsubscriber
   {
```

```
      SwitchSubscriberConfig
      {
         name = "MySubscriber";
         topicNameList =
         {
            "kabira.kts.trace.flow"
         };
         languageId = "";
      };
   };
   configure cfgsubscriber
   {
      SubscriberPropertyConfig
      {
         subscriberName = "MySubscriber";
         name = "filenameTemplate";
         value = "kts_%nodeName_%Y%m%d%H%M%S_%count.log";
      };
      SubscriberPropertyConfig
      {
         subscriberName = "MySubscriber";
         name = "fileOpenMode";
         value = "Append";
      };
      SubscriberPropertyConfig
      {
         subscriberName = "MySubscriber";
         name = "fileAccessPermissions";
         value = "0666";
      };
      SubscriberPropertyConfig
      {
         subscriberName = "MySubscriber";
         name = "formatString";
         value = "%Y-%m-%d

%H:%M:%S|%nanoseconds|%topic|%source|%idString|%originator|%aggregation|%eventString";
      };
      SubscriberPropertyConfig
      {
         subscriberName = "MySubscriber";
         name = "maxFileSizeRollover";
         value = "Enable";
      };
   };
};
```

The following table describes each of the properties that you can set in a SwitchSubscriber configuration:

Property	Description
`transactionAbortAction`	controls the write position in the log file after a transaction is aborted; possible values: • `Rewind` – the file position is rolled back to the position at the start of the transaction • `None` – the file position is not rolled back
`filepathTemplate`	output directory for the log files (e.g., `.`, `../mypath`, `/path/to/logfile`)

Property	Description
filenameTemplate	format for the name of log files (e.g., subscriber1_%nodeName_%Y-%m-%d_%H:%M:%S_%count.log) ; the following placeholders are allowed: • %nodeName – the name of the node on which the engine generating the event is running • %count – rollover counter maintained by SwitchSubscriber (the counter is reset each time a new version of the configuration is activated) • swbuiltin::DateFunctions::formatDate placeholders (%Y-%m-%d-%H:%M:%S)
fileOpenMode	controls mode in which log files are opened (or created) for writing; possible values: • Append – write to end-of-file • Truncate – truncate the log file to zero before writing
mkdirMode	file permissions for output directory creation, if the directory does not exist; an octal number of the form 0nnn (e.g., 0755)
formatString	format of event output (e.g., %Y-%m-%d %H:%M:%S %topic %eventString); the following is a list of the most commonly used placeholders: • %topic – event topic name • %source – name of the catalog that defines the event • %id – fully qualified event Id • %eventString – event description, may not be available • %timestamp – timestamp for event; use • %nanoseconds – high resolution timestamp for event
maxFileSizeRollover	enables/disables file rollover when the log-file size exceeds the maxFileSize threshold (in megabytes); possible values: • Enabled – enables rollover for maxFileSize • Disabled – disables rollover for maxFilesize
maxFileSize	maximum size in megabytes of the log file before a file rollover occurs when maxFileSizeRollover is enabled
maxEventsRollover	enables/disables file rollover when the number of events exceeds the maxEvents threshold; possible values • Enabled – enables rollover for maxEvents

Property	Description
	• Disabled – disables rollover for `maxEvents`
`maxEvents`	maximum number of events before a file rollover occurs when `maxEventsRollovers` is enabled.

Overview of KTP topics

The following table lists the types of events published for the different KTP topics.

Topic	Deals with
`kabira.kts`	all KTP events
`kabira.kts.flow`	flow loading and versioning
`kabira.kts.configuration`	configuration loading and versioning
`kabira.kts.logger`	KTP logger activities
`kabira.kts.scheduler`	KTP scheduler activities
`kabira.kts.administration`	switchadmin command execution
`kts.swperf.errors`	swperf routing errors
`kabira.kts.swpassthru`	swpassthru routing errors
`kabira.kts.channel.endpoint`	channel-based endpoint lifecycle and errors
`kabira.kts.channel.service`	channel-based service lifecycle and errors
`kabira.kts.cfgswitchsession`	switchsession configuration errors
`kabira.kts.switchsession.endpoint`	switchsession-based endpoint lifecyle and errors
`kabira.kts.switchsession.service`	switchsession-based service lifecycle and errors
`kabira.kts.switchsession.session`	switchsession-based session lifecycle and errors
`kabira.kts.switchsession.sessionFact-ory`	switchsession-based session factory errors
`kabira.kts.switchtest`	message tracing at endpointsim instances
`kabira.kts.trace.flow`	message tracing at flows
`kabira.kts.trace.channel.endpoint`	message tracing at channel endpoints
`kabira.kts.trace.switchsession`	message tracing at session and session factory
`kabira.kts.trace.switchsession.ses-sion`	message tracing at session
`kabira.kts.trace.switchsession.ses-sionfactory`	message tracing at session factory
`kabira.kts.trace.transform.fmt`	message tracing at fmt transfrom
`kabira.kts.trace.transform.xmt`	message tracing at xmt transform

5

Managing Engines

This chapter tells you how to manage engines and components installed and running on a Kabira Transaction Platform (KTP) node.

KTP applications are comprised of one or more components running as a set of processes known as engines on a node.

A component is the unit of reuse and deployment. All components have a public interface that defines the services the component provides. Normally, you use the Design Center to generate components from models; however you can implement components using other technologies as well. From the outside, it is both impossible and unimportant to know how the component is implemented.

You assign components to one or more processes, known as engines, that execute on a node. A node is the application service environment for components. The node provides all the runtime services and management infrastructure necessary to execute components (On managing nodes, refer to the guide *Managing Kabira Solutions*.

You use a declarative deploy specification to assign components to engines and configure those engines. Deploy specifications are like installation scripts; they identify the components that make up an application, they specify how those components are assigned to processes, and they define any runtime properties that require special configuration.

Deploy specifications are loaded into a node using when the node is installed using the switchadmin command.

Engines

Engines are the unit of execution for KTP. Each engine maps directly to an operating system process. An engine can host one or more components.

Generally speaking, it is best to combine as many components in the same engine for the best possible performance. This allows the runtime to dispatch events via direct function calls rather than across the event bus to a different process.

However, there are valid reasons why you would not want to combine components in the same engine, for example, when a component:

- makes heavy use of process resources (such as file descriptors) and needs to run separately
- must start and stop independently of other components
- specifically forbids being run with other components in the same process (see the section called "componentType" on page 28 and the section called "componentSingleton" on page 28)

There are many generic engine properties that you may configure. These include path names, executable type, log files, recovery policies, and tracing information, among others.

In addition, components may add specific configuration properties to the engines in which they reside. For example, JSA-enabled components add Java specific properties. Fortunately, all of these properties have sensible defaults and most components will run with no extra configuration. Where specific configuration is required, you can specify those properties in the deploy specification along with the engine definitions (see the section called "Deploy specifications" on page 29).

Once you install an engine, you can view and modify all of its properties with one of the engine management tools.

Components

Building components from models is described in detail in the *KTP Developer's Guide*. You define component properties in the component specification used to build the component. Generally, component properties are only of interest at build time. However, there are a few components properties that directly affect component deployment:

componentType The following table shows different possibilities for componentType and describes deployment aspects:

DYNAMIC	This is the normal component type for KTP components. This property says that the component can be dynamically loaded into an engine and can co-exist with other DYNAMIC components.
STATIC	This is a special case, which says the component cannot be dynamically loaded. Typically this happens when a component must link with third party libraries that do not support dynamic loading. When this property is specified, the Design Center builds a statically linked component that must be the only STATIC or DYNAMIC component assigned to an engine.
LIBRARY	LIBRARY types provide support functions such as the swbuiltin package. They may be assigned to any engine that contains components that require these functions, both STATIC and DYNAMIC.

componentSingleton This property indicates that only one instance of a component can be running on a node. Typically, you want to specify a singleton component if the component stores process resources in attributes and cannot support multiple instances. It is non-deterministic which component instance services an event. Thus, the situation can arise where a process resource created in one engine is handled by a different engine where that process resource is not valid.

Deploy specifications

Deploy specifications define

- engines
- engine properties
- node properties.
- components assigned to engines
- search paths

Creating a deploy specification

The format of deploy specifications is similar to component specifications. Each deploy specification starts with a deploy keyword:

```
deploy MyApp
{
   ...
};
```

The deploy keyword serves several purposes:

- it groups together all engines defined within a deploy block together for administration purposes
- it scopes the names of engines to the deploy block; for example, if the deploy block is called `MyApp` , then administration tools display the names of all engines in the block start with `MyApp::` .
- it allows you to group sets of components together for reuse (see the section called “Exporting and using components and properties” on page 32); this is useful when shipping component frameworks

Defining engines

Within a deploy block, you define engines, with the engine keyword. You can define any number of engines:

```
deploy MyApp
{
    engine EngineA
    {
    };
    engine EngineB
    {
    };
};
```

As stated, engines are scoped to the deploy block, so the fully scoped name of `EngineA` is `MyApp::EngineA` in the engine management tools.

Adding components to engines

Within an engine block, you list the components that you want to assign to the engine with the component keyword:

```
deploy MyApp
{
    engine EngineA
    {
        component Comp1;
        component Comp2;
    };
    engine EngineB
    {
        component Comp1;
        component Comp3;
    };
};
```

When you load a deploy specification, the engine management tools ensures that the deploy specification does not violate any component constraints.

For example, they ensure that a DYNAMIC component has not been placed in the same engine as a STATIC one or that a singleton component does not appear in more than one engine.

Setting component search rules

There are two default search paths where the engine management tools look for components: `"."` and `$SW_HOME/distrib/$SW_PLATFORM/component`. You can provide additional search paths with the importPath property.

```
importPath=/search/path/one;
deploy MyApp
{
    importPath=/search/path/two;
    engine EngineA
    {
        importPath=/search/path/three;
        component Comp1
        {
            importPath=/search/path/four;
        };
    };
};
```

Search paths are processed from innermost scope outward. In the previous example, the search order for `Comp1` would be:

```
/search/path/four
/search/path/three
/search/path/two
/search/path/one
```

When you specify a component that you want to add to an engine, you do not supply a path or file extension.

Setting engine properties

You can provide additional engine configuration properties to override defaults. For example:

```
deploy MyApp
{
    engine EngineA
    {
        buildType=DEVELOPMENT;
        component Comp1;
    };
};
```

You can scope some engine properties to their own blocks. To do this, use the config keyword to specify the block:

```
deploy MyApp
{
    engine EngineA
    {
        config Environment
        {
            envVar=someValue;
        }
        config Java
        {
            classPath=/some/path;
        };
        component Comp1;
    };
};
```

The engine property must be valid; otherwise a load error occurs. See Chapter 7 for a list of the available properties.

Setting node properties

You can also set node properties. Use the config keyword to specify a block within the deploy block. See Chapter 7 for more details about setting node properties.

Using special syntax

You can use either the C or C++ styles for comments:

```
/*  C style
    comment */
// C++ style comment
```

You can also quote special characters or keywords in that appear in properties:

```
deploy MyApp
{
    engine EngineA
    {
        config "config"
        {
            "-arg"="some value";
        }
```

```
        component Comp1;
    };
};
```

Strings can span multiple lines, or split into parts to avoid the newline:

```
deploy MyApp
{
    engine EngineA
    {
        config "config"
        {
            "-arg"="some"
            " value";
            string="this is
                    a multiline
                    string";
        }
        component Comp1;
    };
};
```

Strings can also contain the following special sequences: `\n` , `\t` , `\v` , `\b` , `\r` , `\f` , `\a` , and `\\` .

Using macros

Deploy specifications support macros. Macros substitution takes place before the file is parsed, so you can use macros for anything. Typically, you use them for property values. For example:

```
deploy MyApp
{
    engine EngineA
    {
        buildType=$(BUILD);
        component Comp1;
    };
};
```

Actual values provided by administration tools replace the formal parameters when the macro runs.

Exporting and using components and properties

You may want to ship products as frameworks, designed for use by other programmers. Deploy specifications provide a syntax that allows you to specify a framework as a single unit.

The export keyword groups a set of components and properties that define a framework:

```
deploy MyFramework
{
    export
    {
        property=value;
        component Comp1;
        component Comp2;
    };
};
```

You can then refer to the framework from another deploy specification:

```
importPath=/some/path;
import frmwk;
deploy MyApp
{
    engine EngineA
    {
        use MyFramework;
        component Comp3;
    };
};
```

Typically this is a two step process:

- import the framework's deploy specification
- specify engines that use properties and components that the framework exports

The import statement allows you to insert the contents of the specified deploy specification at the point of the import statement. You specify the name of the file without path and file extension. The system searches for the `.kds` file using the same search path as components. You can use importPath statements to augment the search path, but they must appear before the import statement in the file.

You specify that an engine uses exported properties and components with the use statement. The use statement expects the name of a deploy specification. The system inserts the contents of all export blocks found in the specified deploy specification into the engine that uses it. This includes properties set within the export block. If the engine block sets the same property, the engine block's property setting takes precedence.

There can be multiple export blocks in a deploy block.

The following example illustrates how the system processes import, export, and use statements. Assume there are the following two deploy specifications, in separate files, `frmwrk.kds` and `client.kds` .

File `frmwrk.kds` :

```
deploy MyFramework
{
    export
    {
        name=value1;
        component Comp1;
    };
    engine EngineA
    {
        component Comp2;
    };
};
```

File `client.kds` :

```
import frmwrk;
deploy MyApp
{
    engine EngineB
    {
        use MyFramework;
        component Comp3;
```

```
    };
};
```

Loading `client.kds` results in a deploy specification that you could also write as follows:

```
deploy MyFramework
{
    engine EngineA
    {
        component Comp2;
    };
}
deploy MyApp
{
    engine EngineB
    {
        name=value1;
        component Comp1;
        component Comp3;
    };
};
```

Adding an external engine

You can define an engine that allows the System Coordinator to control the execution of external programs.

Use a deploy specification like the following to create an external engine:

```
deploy YourExternalEngine
{
   engine YourExternalEngine
   {
      component ExternalProgram;
      // Required configuration
      name="Display Name";
      description="Description";
      developmentExecutable=/path/to/executable;
      productionExecutable=/path/to/executable;
      engineArguments="-arg1 -arg2";
      // Optional configuration
      autoStart=(true false);
      restart=(true false);
      needsRecovery=(true false);
      triggersRecovery=(true false);
      maxFailCount=10;
      engineExitValues="0";
      engineKillSignal=15;
      config Environment
      {
      };
   };
};
```

Security management

You can define the security policy for model elements. The security policy lets you specify who can access elements in a component, and what type of access is allowed on each of those elements. You specify this policy using the Kabira configuration service as summarized below and detailed in Chapter 3.

Security policy concepts and features

A security policy specifies the access control rules that apply to roles associated with authenticated principals. These concepts are described below.

Authentication and principals A principal is a user or agent on whose behalf work is to be done in the system. The process of authentication involves verifying the identity of a principal accessing a specific node.

Secured model elements can only be accessed using adapters that provide authentication. One example is the authenticated form of os_connect() in the osw PHP engine.

Principals and roles An authenticated principal can have one or more roles associated with it. A role describes a user type, such as “customer” or “administrator”.

Roles and access control Access to elements in the component is governed by access control rules. A rule grants permission for a role to access an element of the component, and specifies the type of access allowed.

Access permission is always granted to a role, and thus all principals associated with that role. Explicit access permission cannot be granted directly to principals.

Configuring security

You can specify access control rules for the following model elements:

- Interfaces
- Individual attributes and operations within an interface
- Relationships defined for interfaces
- Extents of interfaces—access by select
- Keys with interfaces—access by select ... where <key=value>

You can also specify access control rules on model elements at a higher scope:

- Module—applies to all interfaces in the module
- Package—applies to all interfaces in the package

Distributed discovery management

Distributed applications may use a distributed discovery protocol to identify nodes participating in the application. On multi-homed hosts it is necessary to specify which network interface to use for the discovery protocol.

You specify this in the distribution section of a deployment specification as follows:

```
config Distribution
{
    config DiscoveryService
```

```
    {
        broadcastHost=hostName;
    };
};
```

Where hostName is the host name of the interface that the distribution engine will use for discovery service. The default hostname is the name returned from the gethostname(3C) library call.

Hosts that are not multi-homed do not need a distribution section in their deployment specifications.

Adding, replacing, and removing engines

You deploy an engine by loading the corresponding deploy specification (see the section called "Deploy specifications" on page 29). You can load a deploy specification with either of the engine management tools.

When you deploy an engine for the first time, the coordinator sets the engine's configuration to the default values built into the component archive or those overridden in the deploy specification. It creates a deployment directory for the engine, a subdirectory of the node's runtime path. It extracts all the files necessary for deploying the component from the component archive to the deployment directory.

For dynamic engines (engines containing dynamic components), a load or replace causes a reload to occur. This means that the coordinator unloads all components and then reloads those specified in the deploy specification. This in turn invokes any operations in the model with the [unload] and [reload] properties. All engine settings are retained.

Reloading or replacing a static engine (an engine containing a static component), or changing an engine from dynamic to static (or vice versa), stops and restarts the engine. All engine settings are retained.

When you reload a deploy specification, the components may or may not be compatible with the existing components (see the section called "Type mismatch errors" on page 36). Changes to a component that are compatible with earlier versions include:

- adding new entities and interfaces
- adding triggers
- changes to action language

Incompatible changes to an engine include:

- adding or removing attributes
- changing operation signatures
- adding or removing states or transitions

Type mismatch errors

If you receive a type mismatch error when loading a deploy specification, components being added contain types that conflict with types already in shared memory. You cannot allow this situation to persist; you must remove the corresponding engine.

You can replace an engine with the Force Type Removal option; however, use this with extreme caution, as it removes from shared memory all instances of a type and all events destined for it. Furthermore, it is not a panacea for type mismatch errors; type mismatches will continue to occur if you do not remove the conflicting engines.

If you encounter a type mismatch error, proceed as follows:

1. determine which component contains the object types that conflict (with dynamic engines, the same component may appear in multiple engines; you must reload each one of these engines with the new component, or remove altogether)
2. stop all engines.
3. remove any engines that should no longer exist in the system
4. load the deploy specification for the affected engines using the Force Type Removal option
5. start all engines.

Trace files

KTP engines and the System Coordinator can generate trace messages into log files in the node runtime directory. These messages can be useful for debugging applications or diagnosing system performance. Each engine uses its own trace file, as does the System Coordinator. You can use the engine management tools to set the filename, maximum size, and other characteristics of trace files.

An engine (or the system coordinator) only generates the types of trace messages that it is set to generate. You can use the engine management tools to control the trace level and trace filters used when generating a trace.

For more on configuring trace files and filters, see the section called “Tracing section” on page 60.

Trace severity

There are four trace severities defined in KTP. In decreasing order of severity, they are: fatal, warning, info, and debug. You can set the overall level of tracing for the engine or coordinator.

Generating trace files can have a large negative effect on system performance. Enabling debug tracing will cause serious system performance degradation.

Trace filters

In addition to the overall level of trace messages, you can also configure which parts of KTP can generate each level of trace message.

For each individual engine, and for the System Coordinator, you can set:

- the trace level
- the trace severity

Engine lifecycle events

Engines have the following states:

- stopped - engine was stopped normally by the administrator.
- running - engine is running normally.
- error - engine has detected an error.
- terminating - engine is stopping normally.
- killing - engine did not shut down cleanly. It is now stopping abnormally.

Engines can be reloaded. An engine reload pauses engine execution, unloads all components, reloads the components, and restarts execution without exiting the engine operating system process. This allows an engine to hold open all process level resources like file descriptors and sockets during a component upgrade. The operations called for engine lifecycle changes are as follows:

- Engine Started - [packageinitialize] and [initialize] called
- Engine Reloaded - [unload], [packagereload] and [reload] called
- Engine Stopped - [terminate] called

Details on the IDLos model properties to identify operations as engine lifecycle events are below.

[packageinitialize]

The system invokes `[packageinitialize]` operations when an engine starts, but before events are dispatched. Thus, such operations are guaranteed to execute before the engine starts processing other work. However, `[packageinitialize]` operations are restricted to two-way operations, without parameters and a return value, to entities in the same package. A runtime error is raised if these operations attempt to call an interface in another package.

[packagereload]

The system invokes `[packagereload]` operations when an engine is reloaded, but before events are dispatched. Thus, such operations are guaranteed to execute before the engine starts processing other work. However, `[packagereload]` operations are restricted to two-way operations, without parameters and a return value, to entities in the same package. A runtime error is raised if these operations attempt to call an interface in another package.

[initialize]

The system invokes `[initialize]` operations when an engine starts, but after the event bus starts. Thus, their invocation may occur after the engine already begins to process work. This limits their usefulness for initialization. However, `[initialize]` operations do not suffer the same restrictions as `[packageinitialize]` operations. The initialize operation is not called after a reload.

[unload]

The system invokes `[unload]` operations before an engine unloads its components. You can use `[unload]` operations to free any process resources.

[reload]

The system invokes `[reload]` operations when an engine has completed reloading its components. You can use `[reload]` operations to reinitialize process resources.

[terminate]

The system invokes `[terminate]` operations during ordinary engine shutdown. It is not called when an engine exits with an error or after a reload. The reload operation is called in the case of a reload.

6

Kabira Monitor

The Kabira Monitor is a powerful diagnostic tool that allows you to interactively decode and view any data structure in shared memory. For example it is possible to display user objects in shared memory making it extremely helpful in monitoring and debugging applications that you develop with KTP.

Starting the Kabira Monitor

You start the Kabira Monitor by entering the following command at a shell prompt:

```
swmon [shared memory filename]
```

Your path should be set correctly before you use KTP commands. See “User Environment” on page 3

Specifying a shared memory file in the command line is optional; you can attach a shared memory file after launching the Kabira Monitor.

Working with the Kabira Monitor

The Kabira Monitor display is divided into two panes. The left pane is a “table of contents,” or index, of all the objects in the system that apply to the current view. The right pane displays the contents of the currently selected object.

The entities and extents that are listed in the left pane function as object containers. This is indicated by the toggle icon preceding the respective identifier. By clicking on the corresponding toggle icon, you can expand and collapse a container; this displays and hides the object references of all objects of that type in shared memory.

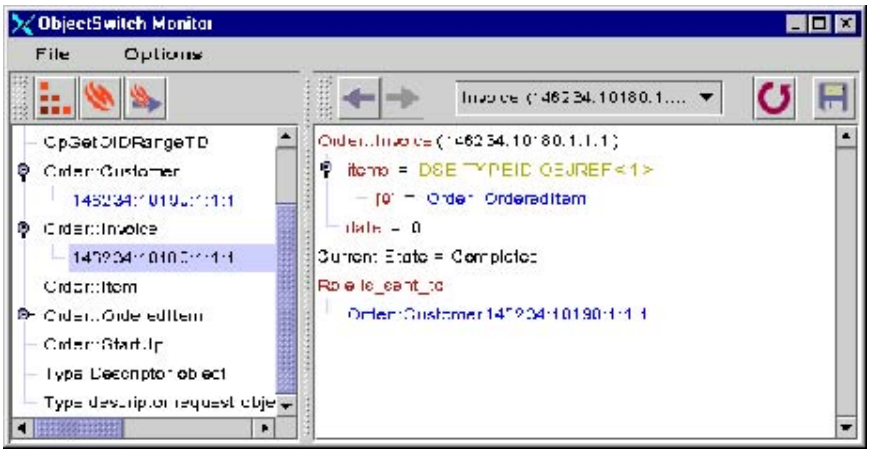

Figure 6.1. Kabira Monitor

Color coding

The following table describes the color coding used to represent the different types of elements:

Color	Type
blue	offset or object reference
gold	type reference
red	slot in an object's shared memory image
green	string
black	literal value or primitive type

File menu

Provides commands for managing files associated with the Kabira Monitor as well as for closing the Kabira Monitor.

Attach Memory... lets you to attach a shared memory file; a file selection dialog opens allowing you to select a shared memory file, typically named "ossm".

Properties... displays the fully qualified name of the shared memory file currently attached to the Kabira Monitor.

Open Log File... selects a file for logging the contents of the right pane of the Kabira Monitor. (See the section called " Writing to a log file " on page 44.)

Close Log File unselects the log file, if one is selected.

Close exits the Kabira Monitor.

Options menu

The commands in the Options menu let you show/hide detail information and set the level of decoding as well as specify the order in which the types are displayed.

Filter Generated Types filters out objects that do not correspond to entities from the user model.

Show Type ID displays types together with their type ID. (See "Type ID and object reference" in the following section for a detailed description of type IDs.)

Show Cardinality displays types together with the number of corresponding objects found in shared memory, which is very helpful in determining the number of instances of a type in shared memory.

Decoding Level... enables the interpretation of different data structures. There are three levels of decoding: Model, System, and Primitive.

- Model decoding—shows object instances as close to the modeled objects as possible; this decoding level will generally be the most valuable in viewing user objects.
- System decoding—displays object instances in terms of their system runtime data structures; however, still automatically decodes certain complex types like strings and arrays.
- Primitive decoding—only interprets primitive types; provides no decoding of complex types.

Sort... lets you pick how you want to sort the types in the left pane

- By Name—sorts alphabetically.
- By Type ID—sorts according to type IDs.

Perferences allows you to set font preferences for the monitor that are saved so that they are in affect the next time the monitor is run.

Buttons—Views

The Kabira Monitor offers three different views of the data structures found in shared memory: the Allocator, Model and Transaction views. Those views correspond to the three buttons above the left pane of the Kabira Monitor:

The Allocator View displays all allocated system objects. A system object is used to support the operation of KTP. It is not a modeled object. The System View is geared towards the Kabira field or support engineer. As an application developer, you will probably not be interested in the data structures presented in this view.

The Model View displays user objects in shared memory, which are generated by KTP from the user model, for example, entities, extents, typedefs, and structs. This view will be of great assistance in monitoring your applications.

The Transaction View is only marginally of interest for a running application; normally, by the time you click on a transaction, it's already complete. This view is intended for developers diagnosing hung systems.

Navigating shared memory

Offsets and object references as well as type references are all represented in the right pane as hotlinks; clicking on one displays the object in the right pane.

You can quickly navigate to related objects by just clicking on the object references of the respective roles and attributes; or traverse type descriptor definitions by clicking on the various type references.

History buttons and list

The Kabira Monitor records a history of the types and instances that it previously displayed. You can navigate the history using the Prior and Next buttons and the drop-down history list, all of which appear above the right pane of the Kabira Monitor.

You can use the

Prior and Next buttons to navigate through the history of previously displayed types and instances.

Alternatively, you can use t

he drop-down history to select a particular type or instance that was previously displayed. Clicking on the control drops down a scrollable history, and clicking on one of the items in the history opens the item's decoded representation.

Reloading the shared memory file

The types and objects displayed by the Kabira Monitor represent a snapshot of shared memory at the instance of navigating to the currently displayed type or object. You update the display with the current state of shared memory by clicking on the Reload button, by clicking on a view button, or clicking on another type or object reference.

Writing to a log file

You can write the contents of the right pane to a log file with the Save button, appending the current contents to the log file. If a log file is not currently selected, clicking on the Save button opens a file selection dialog, allowing you to select a log file before saving. (See the section called " Open Log File... " on page 42.)

Viewing types and instances

The left pane of the Kabira Monitor lists the names of all the types that apply to the current view. When shared memory contains objects of a particular type, the Design Center lists references to those objects (object IDs) under the name of the type and displays a toggle icon in front of the type's name. Clicking on the toggle icon shows and hides the object references of that type.

Figure 6.2 illustrates the Model View where the `Order::Customer` and `Order::Invoice` types have been expanded to show references to objects of those types.

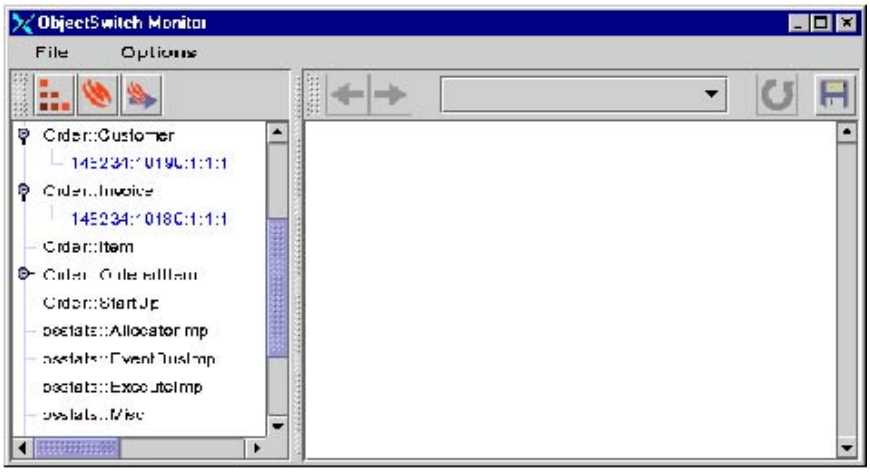

Figure 6.2. Types and object references in the left pane

Type ID and object reference

A type ID is used to identify each described type. These type IDs are based on the fully qualified type name.

Object references are based on type IDs, and uniquely identify an instance throughout all KTP nodes. An object reference is made up of four parts which are described below.

type ID	implementation	location	OID

The type ID is a two-part construction, usually given in the form `dddd:tttt` so that the Kabira Monitor displays these fields in the form:

```
dddd:tttt:iiii:llll:oooo
```

Type ID This is generated by the object compiler. The type ID is unique within a repository and identifies a KTP type.

Implementation This is the object implementation—either transient or persistent, using some implementation adapter. It will be in the range 1-4, where:

1	The default KTP transient implementation.
2	Sybase persistent implementation.
3	Informix persistent implementation.
4	Oracle persistent implementation.

Location This is the location code of the KTP node where an instance is created.

OID This is generated at runtime, and only has to be unique within the scope of all the above qualifiers. For persistent objects that are uniquely keyed with a numeric value, it is generated by the backing database. For all other objects, KTP generates the OID.

Decoded Objects

When you click on an object reference, the Kabira Monitor displays the decoded representation of that object in the right pane, which can include:

- Attributes with their current values
- Roles with references to the related objects
- The current state

The attributes can contain nested values, a sequence, struct, or array, which you can expand and view member or elements by clicking on the toggle icon.

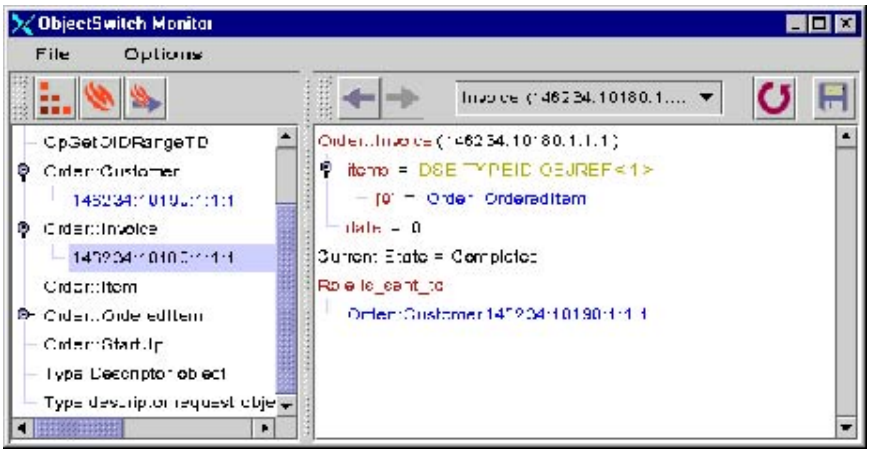

Figure 6.3. Decoded objects

Current state If a state machine is defined for an object, the Kabira Monitor displays the object's current state. Now by definition of a state machine (see the "State machines" section of the "Developer's Guide"), an action is executed upon entering each state. However, the current state is not updated until the corresponding transaction has been committed.

Thus, if you halt execution during an object's state action and view the decoded representation of that object in the Transaction view, the displayed current state is not the state being entered but the last committed state.

Decoded Types

KTP maintains type information in shared memory using a type descriptor object to describe each type. Type descriptors are meta-data, and as such can describe the structure of any object in shared memory. Each such object describes in a generic fashion how the objects of the respective type are constructed. Every type in the system has a type descriptor.

When you select a type name in the left pane (or right pane) of the Kabira Monitor, the type descriptor object for that type is displayed in the right pane:

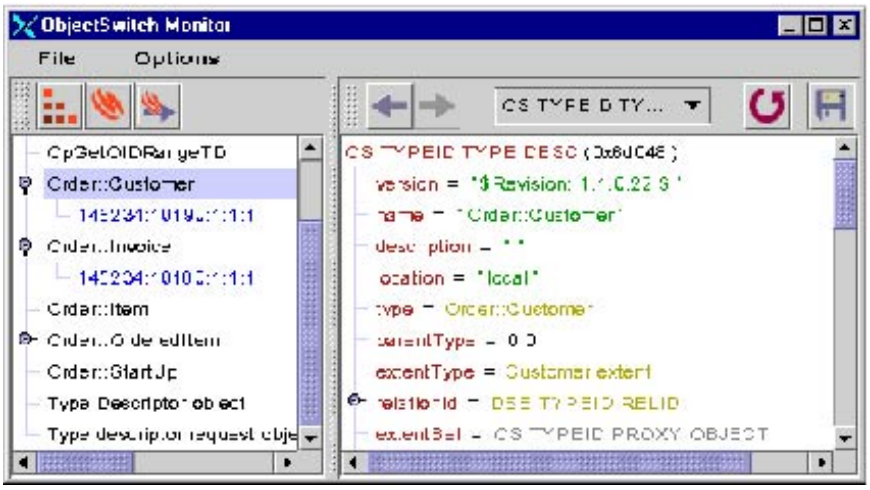

Figure 6.4. Type information

Each type descriptor objects has, for example, slots for version, name, description, type, parent type (inheritance), extent, size and species (struct, exception, dynamic, sequence, array, or interface). Only the values assigned to the different slots vary for each type. The entry `objectSize` can be used to determine the size of that object in shared memory.

Some slots reference a nested type, a struct, sequence, or array. The Kabira Monitor indicates that the nested type contains values that can be viewed by preceding the slot identifier with a toggle icon. Clicking on the toggle icon expands the slot to display underneath the identifier the contents of the referenced type. The nested slots may contain still other expandable type references.

If the nested type is a sequence, the Kabira Monitor displays the type reference with following angle brackets (“<>”). The Kabira Monitor shows the number of elements contained in the sequence inside the angle brackets.

Clicking on any slot identifier in the right pane shows/hides the technical details of that slot. Figure 6.5 shows the expanded `type` slot identifier:

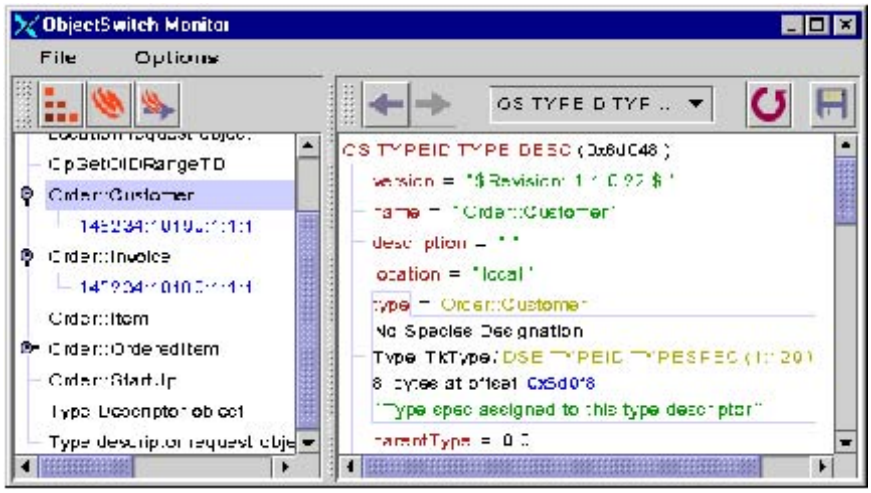

Figure 6.5. Slot details

Slot details include:

- species (an enumeration of the “kind” of object)
- type ID (with a hotlink to the type descriptor)
- slot size
- physical memory address
- slot description

An example model view

This section introduces an example model and illustrates how the Kabira Monitor represents the types, objects, and transactions.

The example model

The model consists of two main entities, `Invoice` and `Customer`, which are related through relationship `R1`.

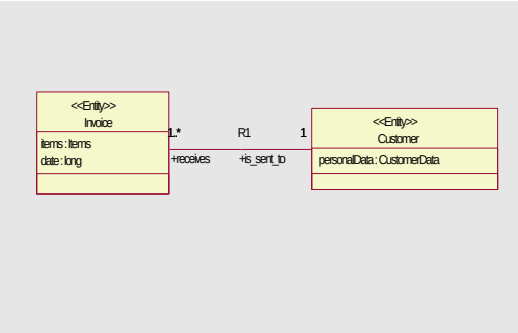

Figure 6.6. The general user model

Invoice The items included in an `Invoice` object, represented by the `items` attribute, are contained in a sequence of `OrderedItem`.

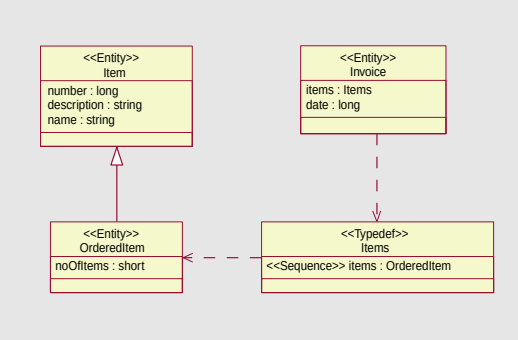

Figure 6.7. Invoice

`OrderedItem` is a specialized type of the `Item` entity, additionally giving the quantity for each specific item being ordered. The date of the `Invoice` object is given by the `date` attribute.

As shown in Figure 6.8, an `Invoice` object has 5 different states: `OrderTaken` , `Billing` , `Completed` , `Cancelled` , and `PoliceNotified` . There is a transition from the initial state `OrderTaken` to `Billing` when an invoice receives the `billCustomer` signal, which requires two arguments, one of type `Items` , a sequence or `OrderedItem` , and the other an object of type `Customer` . The action executed when entering the `Billing` state performs a credit check on the customer and depending on the outcome, sends a signal `goodCustomer` , `badCustomer` , or `uglyCustomer` , which causes a transition to either the `Completed` , `Cancelled` , or `PoliceNotified` state, respectively.

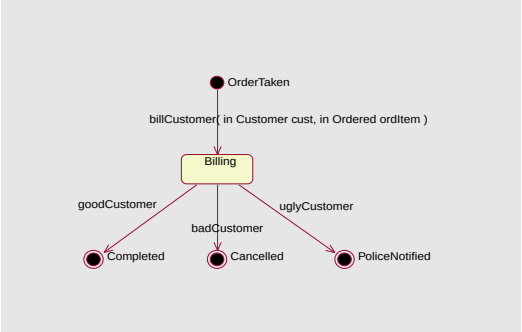

Figure 6.8. The Invoice state model

Customer The `Customer` entity contains an attribute defined as a struct `CustomerData` , which specifies the personal data for one customer, including a member defined with the help of an enumeration, `CustomerType` , specifying the type of customer, `good` , `bad` , or `ugly` .

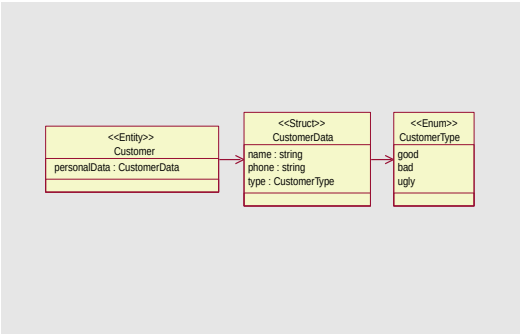

Figure 6.9. Customer

You can create an `OrderedItem` object and add it to a sequence of `OrderedItem` objects with action language like the following:

```
declare Items ordItems;
declare OrderedItem ordItem;
create ordItem;
ordItem.number = 01234;
ordItem.description = "Six-string electric guitar...";
ordItem.name = "Fender Stratocaster";
ordItem.noOfItems = 1;
ordItems[ordItems.length] = ordItem;
```

The following action language code creates a new customer:

```
// John's data
declare CustomerData johnData;
johnData.name = "John Lennon";
johnData.phone = "415/567-3425";
declare Customer john;
create john;
john.personalData = johnData;
```

An `Invoice` object is created with the next action language statements:

```
declare Invoice inv;
create inv;
```

When you create an `Invoice` object, it goes into the `OrderTaken` state and awaits the `billCustomer` signal, to bill a customer for a number of ordered items. For example, the customer `john` is billed for some ordered items `ordItems` with the following code:

```
inv.billCustomer(john, ordItems);
```

The `Invoice` object goes into the `Billing` state and executes the following action language, acting on the `cust` argument, of type `Customer` , and the `ordItems` argument, of type `Items` :

```
declare ::Order::CustomerData custData;
declare ::Order::Customer localCust;
localCust = cust;
custData = localCust.personalData;
declare ::Credit::CreditRating crdrating;
create crdrating;

// Stops execution for 20 seconds
// allowing the current transaction to be viewed.
SWBuiltIn::nanoSleep(20,0);
crdrating.checkRating(custData.name);
localCust.rating = crdrating;

if (crdrating.rating == 0)
{
    custData.type = good;
    self.items = ordItems;
    relate localCust receives self;
    relate self is_sent_to localCust;
    self.goodCustomer();
}
else if (crdrating.rating ==1)
{
    custData.type = bad;
    self.badCustomer();
}
else
{
    custData.type = ugly;
    self.uglyCustomer();
}
```

A credit check is performed on the customer using the customer's name and the customer type is updated accordingly.

- If the credit check is good (= 0), the ordered items are assigned to the invoice, the relationship is set up between the invoice and the customer, and with the `goodCustomer` signal, the invoice goes into the `Completed` state.
- If the credit check is bad (= 1), the invoice goes into the `Cancelled` state with the `badCustomer` signal.
- Otherwise, the invoice goes into the `PoliceNotified` state with the `uglyCustomer` signal.

The action language additionally contains a built-in function, `nanoSleep`, that stops execution for 20 seconds to provide a glimpse of the Transaction View.

Viewing the model

Attaching the shared memory file to the Kabira Monitor immediately after starting the model application and clicking on the Transaction View produces a Kabira Monitor display similar to the one appearing in Figure 6.10. The presence of an engine instance indicates that execution of this engine has been stopped, in this case due to the built-in function `nanoSleep`.

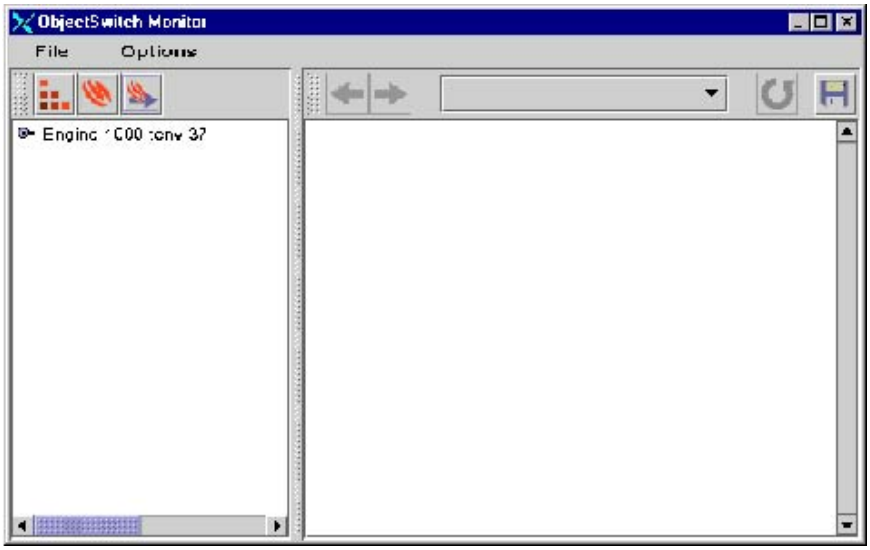

Figure 6.10. Transaction View

Clicking on the toggle icon in front of the engine instance expands the transaction container. The display in the left pane lists an object reference for each object created within the current transaction as well as their extent. Clicking on the object reference for the `Invoice` object displays the decoded representation of that object, as shown in Figure 6.11.

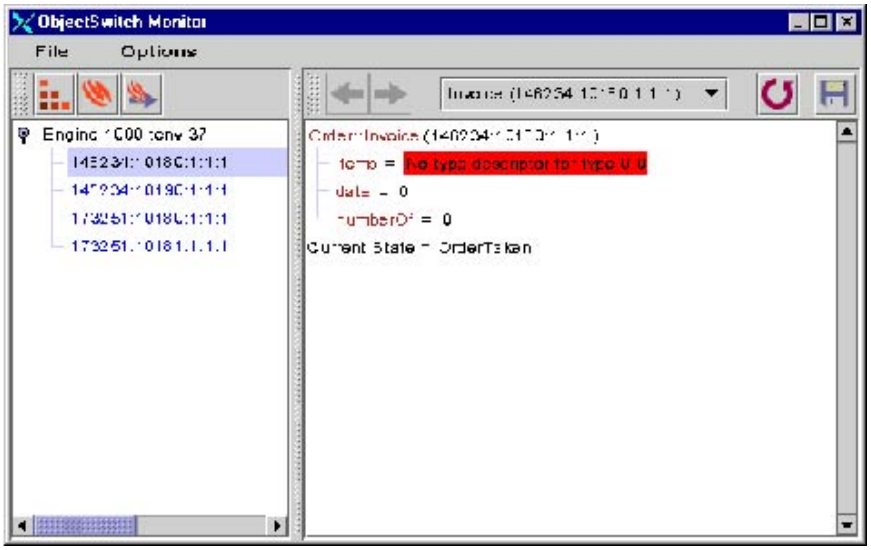

Figure 6.11. Invoice object in stopped transaction

Since the `nanoSleep` function stops execution before any attribute value assignments, you see the default attribute values. The current state is also displayed.

After 20 seconds execution continues and because customer `john` has a good credit rating,

- the sequence of `OrderedItem` , `ordItems` , is assigned to the `items` attribute of the `Invoice` object
- the `Invoice` object is related to the `john` object via the `R1` relationship, and vice versa
- the type attribute of the customer `john` is set to `good`
- there is a transition of the `Invoice` object's state from `Billing` to `Completed`

You can view this from the Model View after clicking on the corresponding button. Clicking on the toggle icon in front of the `Invoice` type expands its contents and clicking on the `Invoice` object reference under the Invoice type shows the decoded representation of the object, as in Figure 6.12.

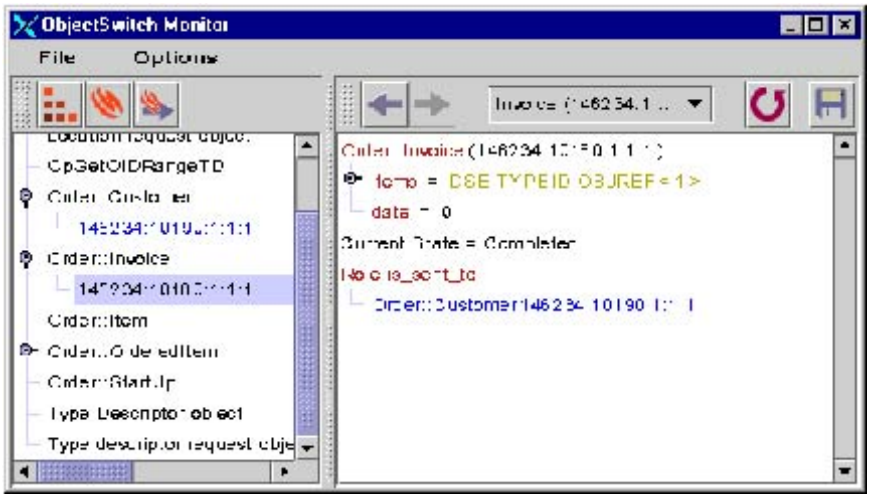

Figure 6.12. Invoice object in the Completed state

Since the `items` attribute is a nested type, a sequence of `OrderedItem` , there is a toggle icon in front of it. Clicking on the toggle icon expands the sequence so that you can view the individual elements, as shown in Figure 6.13.

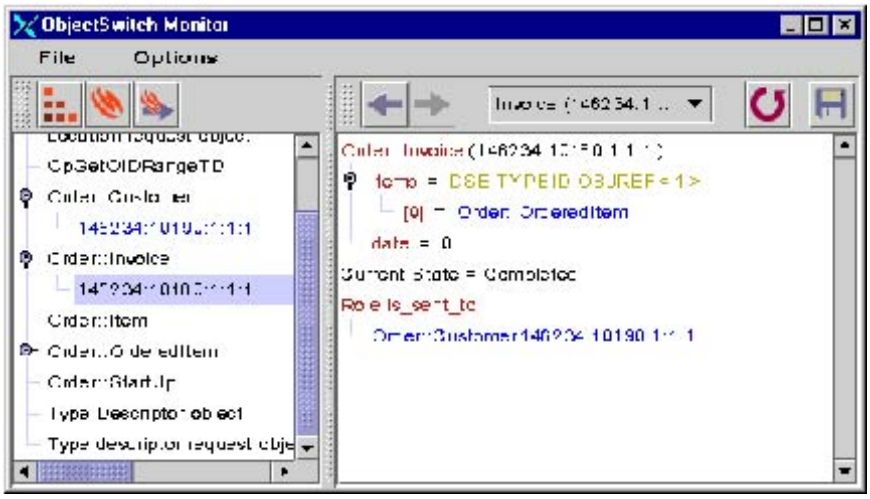

Figure 6.13. Invoice object with expanded items

The sequence contains only one element, which is represented by the hotlinked object reference. Clicking on the object reference opens the decoded representation of the `OrderedItem` object.

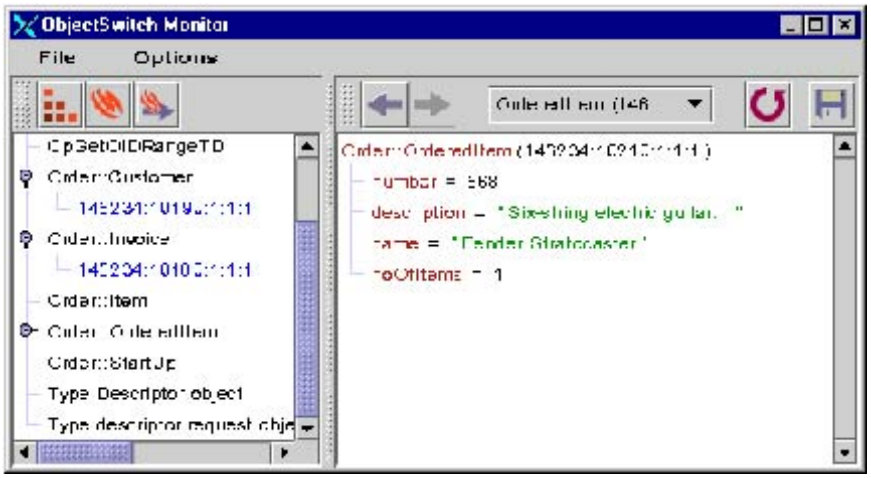

Figure 6.14. OrderedItem object

Clicking on the Previous button returns to the representation of the `Invoice` object displayed in Figure 6.13. The object reference under the `is_sent_to` is a hotlink to the `Customer` object `john` created above.

Clicking on the object reference under the `is_sent_to` role opens the decoded representation of that object in the right pane, as shown in Figure 6.15.

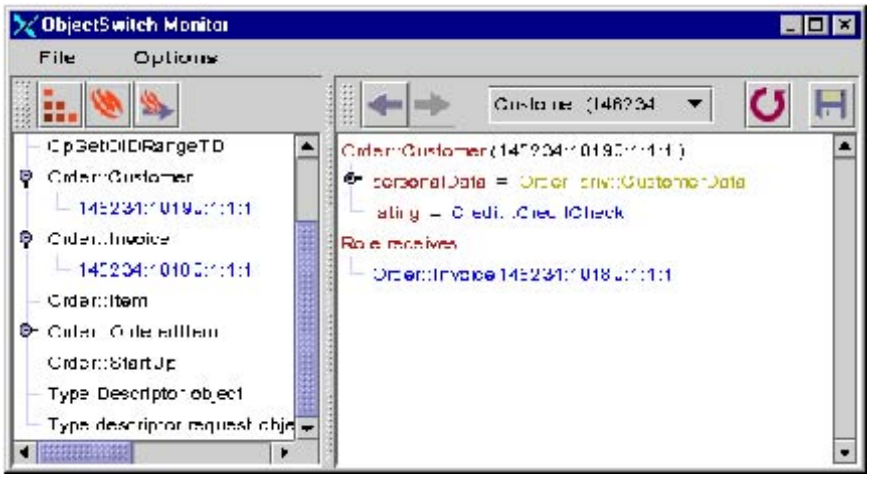

Figure 6.15. Customer object

Clicking on the toggle icon preceding the `personalData` slot identifier expands the slot to display the contents of the nested type.

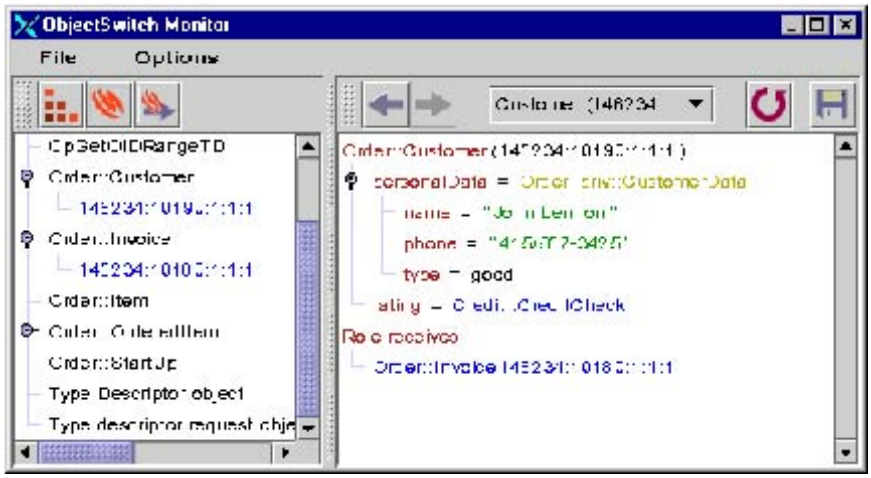

Figure 6.16. A customer object with expanded struct

The object reference appearing underneath the `receives` role in Figure 6.16 is a hotlink back to decoded representation the `Invoice` object. Clicking on the Prior button returns to decoded representation of the `Invoice` object in the right pane, too. Likewise, you can use the drop-down history, as shown in Figure 6.17, to return to the representation of the `Invoice` object displayed in Figure 6.13.

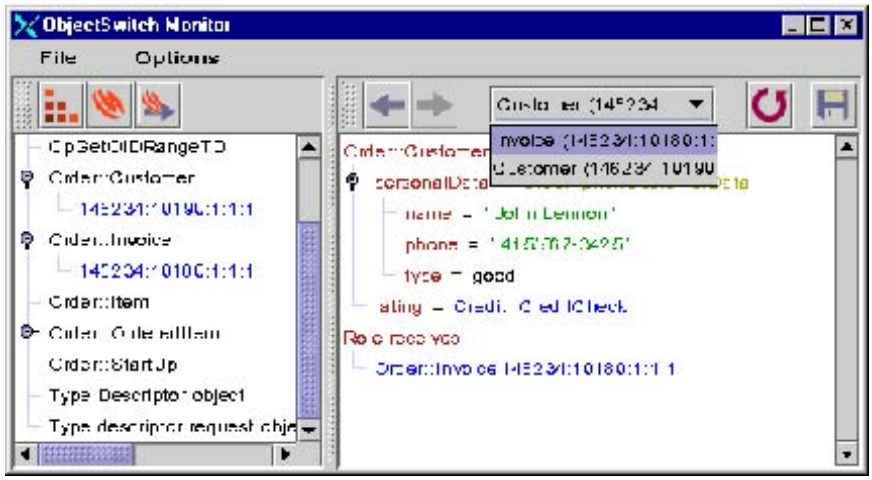

Figure 6.17. An invoice object

Finally, clicking on the `Order::OrderedItem` type reference in the left pane opens the decoded type, as shown in Figure 6.18. Notice the contents of the `parentType` slot, which indicates that `OrderedItem` is a descendent of `Item` .

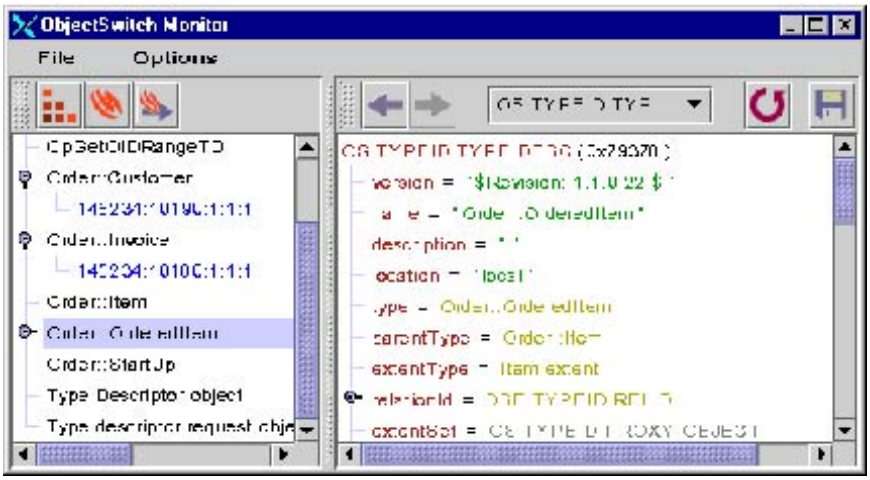

Figure 6.18. The OrderedItem type

7

Deploy specifications

Deploy specifications define how KTP engines are instantiated. This chapter describes the contents of a Kabira deploy specification. Each section rpresents a particular part of the deploy specification.

General section

The following table describes the engine properties in the General section:

Deploy spec element	Description
name	The name displayed in the Engine Control Center. You can only change the value through the corresponding deploy specification.
description	Description information displayed by management tools

Command Line section

The following table describes the engine properties in the Command Line section:

Deploy spec element	Description
buildType	The executable type to execute. This is either DEVELOPMENT, PRODUCTION, or ALL.
developmentExecutable	The path to the development executable file for the engine. This is automatically configured when you add an engine. You should not edit this property.
productionExecutable	The path to the production executable file for the engine. This is automatically configured when you add an engine. You should not edit this property.

Deploy spec element	Description
engineArguments	Arguments passed to the executable on its command line when the process starts. This is automatically configured when you add an engine. You should not edit this property.

Process Control section

The following table describes the engine properties in the Process Control section:

Deploy spec element	Description
autoStart	True if this engine should be started when the coordinator starts up.
restart	True if this engine should restart after it is stopped to perform node recovery.
reload	True if this engine should be reloaded if modified during a deployment specification load.
maxFailCount	Number of consecutive times an engine can fail before the System Coordinator will no longer restart it.
killTimeout	Time to wait for voluntary thread shutdown before thread kill is issued.
abortTimeout	Time to wait for graceful engine shutdown before forced termination.
engineKillTimeout	This is a global limit on the time to wait for any engine to voluntary shutdown. KTP engines have additional properties to control engine termination. See killTimout and abortTimeout. This value should be larger than the abortTimeout set for any engine.
exitOnSystemException	Indicates whether to exit an engine if an unhandled system exception is detected. A value of no indicates that the engine should not be shutdown, a value of yes indicates that the component should be shutdown and node recovery performed. If this property is configured to have a value of no, the runtime will catch the system exception and discard the event that caused it to occur.
lowMemoryWarning-Level	A warning will be written to the engine log file when the shared memory detects that this percentage of shared memory has been used. The message is only printed once per invocation of an engine.
flushRate	The flusher time interval in seconds. The flusher wakes up every flusher time interval to look for objects to flush. A value of 0 disables the flusher.
numTypesPerFlush	This is the number of types the flusher will look at for flush candidates every time it wakes up. If this

Deploy spec element	Description
	number if less than the total number of types installed on the node, the flusher will sequentially walk through all of the types before starting over.
debugPause	Whether to pause the engine in startup after the dlopen of all the libraries, but before any of the initialization code has been run. Useful for attaching a debugger before the engine starts running.

Environment section

The following table describes the engine properties in the Environment section:

Deploy spec element	Description
workingDirectory	Engine working directory. Path must be absolute.
inheritEnvironment	Does the engine inherit the coordinator's environment?
Environment / environment\<variablename>	Environment variables for engine. If you specify an environment variable in this property, it replaces that variable in the parent environment.

Thread Pool section

The following table describes the engine properties in the Thread Pool section:

Deploy spec element	Description
numReadChannels	Initial thread pool size on engine startup. The runtime starts this many threads when the engine is started.
minReadChannels	Minimum thread pool size. The number of threads running in this engine never decreases below this number.
maxReadChannels	Maximum thread pool size. The total number of threads in the engine's thread pool never increases beyond this number.
readChannelInc	Thread pool growth increment. When the thread pool is grown, it always grows by this number.
servicePeriod	Interval time, in seconds, to poll thread pool for starvation. If thread starvation is detected, the thread pool is grown by the pool increment amount.

Transaction section

The following table describes the engine properties in the Transaction section:

Deploy spec element	Description
deadlockBackoffMSec	Time, in milliseconds, to delay transaction replay in the event of a deadlock.

Tracing section

The following table describes the engine properties in the Tracing section:

Deploy spec element	Description
traceEnable	Enable or disable engine tracing. Excessive tracing is detrimental to engine performance.
traceFile	Path to engine trace file.
traceTruncate	Whether to truncate the trace file when the engine starts up.
traceSize	Maximum trace file size before truncation. A value of 0 indicates that the trace file should not be truncated.
traceDebugFilter	The tracing section ends with a long list of individual tracing categories and settings for each: debug, info, warning, and fatal. These settings let you enable or disable trace message generation by category and severity.

Index

U

W

X